yAudit RAMSES Review

Review Resources:

- RAMSES documentation

Auditors:

- Jackson
- prady
- engn33r

Table of Contents

Review Summary

RAMSES

RAMSES is a ve(3,3) DEX that builds on Solidly, but with many improvements. RAMSES v1 has managed millions in TVL, and the new v2 contract files improve upon the existing protocol. The primary upgrade is the staking feature, with much of the other code derived from Uniswap v3.

The contracts of the RAMSES Repo were reviewed over 35 days. The code review was performed by 3 auditors between June 26 and July 30, 2023. The repository was under active development during the review, but the review was limited to the latest commit at the start of the review. This was commit a850f39fd9b0b4c62af5bf4f4d2089284e75cf8e for the RAMSES Exchange repository, and the review scope was focused on the v2, v2-periphery, v2-staking, and v2-universal-router directories.

Scope

The scope of the review consisted of the contracts in the following directories:

- contracts/v2/*
- contracts/v2-periphery/*

- contracts/v2-staking/*
- contracts/v2-universal-router/*

After the findings were presented to the RAMSES team, fixes were made and included in several PRs.

This review is a code review to identify potential vulnerabilities in the code. The reviewers did not investigate security practices or operational security and assumed that privileged accounts could be trusted. The reviewers did not evaluate the security of the code relative to a standard or specification. The review may not have identified all potential attack vectors or areas of vulnerability.

yAudit and the auditors make no warranties regarding the security of the code and do not warrant that the code is free from defects. yAudit and the auditors do not represent nor imply to third parties that the code has been audited nor that the code is free from defects. By deploying or using the code, RAMSES and users of the contracts agree to use the code at their own risk.

Code Evaluation Matrix

Category	Mark	Description
Access Control	Good	Every function that should be protected has proper access control modifiers.
Mathematics	Medium	In most cases the Uniswap libraries are used for mathematical operations. However, in some cases raw mathematical operations are used in lieu of the Uniswap libraries or the dependency LowGasSafeMath libraries.
Complexity	High	Most of the code is forked from Uniswap, however, the new position update logic is new, critical, and highly complex.
Libraries	Good	Most of the new code uses Uniswap and OpenZeppelin libraries where appropriate.
Decentralization	Medium	The protocol developers have roughly the same control as the Uniswap protocol developers. However, RAMSES

Category	Mark	Description
		contracts are upgradeable, whereas Uniswap's are not.
Code stability	Low	Changes were made to the repository while the audit was on-going.
Documentation	Medium	There is documentation explaining the goals of the protocol however, some of the documentation is outdated and no longer reflects the behavior of the current implementation.
Monitoring	Medium	The core interfaces that users will interact with were forked from Uniswap, which emit events where expected. However, there are places in the new code that do not emit events as expected.
Testing and verification	Low	Very little additional testing was added to the new code paths aside from happy case path tests.

Findings Explanation

Findings are broken down into sections by their respective impact:

- Critical, High, Medium, Low impact
  - These are findings that range from attacks that may cause loss of funds, impact control/ownership of the contracts, or cause any unintended consequences/actions that are outside the scope of the requirements.
- Gas savings
  - Findings that can improve the gas efficiency of the contracts.
- Informational
  - Findings including recommendations and best practices.

Critical Findings

None.

High Findings

1. High - Minting with different position indices yields a higher total `boostAmount` than minting with the same index

Technical Details

When minting or burning the `liquidityDelta` is added to an existing position's `liquidity` to determine the `newLiquidity` which is used throughout `_updatePosition()` for various calculations, to include the `veRamRatio` and `newBoostedLiquidity`. Eventually a `boostedLiquidityCap` is calculated using the `veRamRatio` and `hypotheticalLiquidity`. This `boostedLiquidityCap` is intended to limit the amount of `newBoostedLiquidity` that can be added by a position. However, this can be bypassed by using many small positions rather than the same position. This can be done relatively simply by an attacker by using a different index for their positions, and since a `veRamTokenId` balance can be re-used by multiple positions, there is no detriment to using multiple positions to mint liquidity.

See this POC illustrating the point in code:

```
contract AttachTest is Test {

    RamsesV2Pool _pool;
    RamsesV2Factory _factory;
    MockVoter _voter;
    MockNFPManager _nfpManager;
    MockVotingEscrow _veRam;
    MockERC20 _token0;
    MockERC20 _token1;

    function setUp() public {
        _pool = new RamsesV2Pool();
        _factory = new RamsesV2Factory();
        _voter = new MockVoter();
        _nfpManager = new MockNFPManager();
        _veRam = new MockVotingEscrow();
        _token0 = new MockERC20("0", "token_0", 18);
        _token1 = new MockERC20("1", "token_1", 18);
```

```
        deal(address(_token0), address(this), type(uint256).max);
        deal(address(_token1), address(this), type(uint256).max);

        _factory.initialize(address(_nfpManager), address(_veRam), address(_voter),
address(0));
        _pool.initialize(address(_factory), address(_nfpManager), address(_veRam),
address(_voter), address(_token0), address(_token1), 10_000, 200);
        _pool.initialize(3934959759011022346558 29);
    }

    function ramsesV2MintCallback(
        uint256 amount0Owed,
        uint256 amount1Owed,
        bytes calldata data
    ) external {
        _token0.transfer(address(_pool), amount0Owed);
        _token1.transfer(address(_pool), amount1Owed);
    }

    function positionHash(
        address owner,
        uint256 index,
        int24 tickLower,
        int24 tickUpper
    ) internal pure returns (bytes32) {
        return keccak256(abi.encodePacked(owner, index, tickLower, tickUpper));
    }

   function testMultipleAttachmentPOC() public {
        uint256 period = block.timestamp / (1 weeks);
        (
            uint160 sqrtPriceX96,
            int24 tick,
            uint16 observationIndex,
            uint16 observationCardinality,
```

```
            uint16 observationCardinalityNext,
            uint8 feeProtocol,
            bool unlocked
        ) = _pool.slot0();
        int24 tickLower = tick - 132;
        int24 tickUpper = tick + 68;

        // get the boostAmount when using the same index to repeatedly mint
        for (uint i = 0; i < 10; i++) {
            _pool.mint(address(this), 0,  tickLower, tickUpper, 100, 1, "");
        }
        (uint128 boostAmountMintingSameIndex,,,) = _pool.boostInfos(period,
positionHash(address(this), 0, tickLower, tickUpper));
        // clear out all the state from this minting such that it doesn't affect the
minting below.  In essence, the two operations should functional independently of one
another.
        _pool.burn(0, tickLower, tickUpper, 100 * 10, 1);
        _pool.collect(address(this), tickLower, tickUpper,
uint128(_token0.balanceOf(address(_pool))), uint128(_token1.balanceOf(address(_pool))));
        vm.startPrank(address(_pool));
        _token0.transfer(address(this), _token0.balanceOf(address(_pool))); // clear the
fee balances
        _token1.transfer(address(this), _token1.balanceOf(address(_pool)));
        vm.stopPrank();

        skip(1 weeks); // skip forward a week to ensure the prior period's
`totalVeRamAmount` doesn't affect this period's.
        period = period + 1;

        // get the boostAmount when using different indices to repeatedly mint
        uint128 boostAmountMintingDifferentIndices;
        for (uint i = 0; i < 10; i++) {
            _pool.mint(address(this), i+1,  tickLower, tickUpper, 100, 2, ""); // the
VotingEscrow contract is hard coded to return the same balanceOfNFT for every tokenId to
illustrate the point.
            (uint128 boostAmountAccumulator,,,) = _pool.boostInfos(period,
```

```
positionHash(address(this), i+1, tickLower, tickUpper));
            boostAmountMintingDifferentIndices += boostAmountAccumulator;
        }

        // the boostAmount of all the differently indexed positions is > the boost amount
when using the same position supplying the same liquidity
        assertGt(uint(boostAmountMintingDifferentIndices),
uint(boostAmountMintingSameIndex));
    }
}
```

Note that the pool in the POC has been initialized similarly to the deployed DEUS/USDC pool which can be found here. Also, note that the `MockVotingEscrow` contract has been hard coded to return the same `balanceOfNFT()` (the balance taken from the deployed veRAM contract for `tokenId` 1 at the time of the test) and `isApproverOrOwner()` for ease of testing and to illustrate the difference between the two minting strategies.

```
    function isApprovedOrOwner(address, uint256) external override view returns (bool) {
        return true;
    }

    function balanceOfNFT(uint256) external override view returns (uint256) {
        return 21487469130353205716474214;
    }
```

Impact

High. An attacker can ensure that their `newBoostedLiquidity` always falls under the `boostedLiquidityCap`.

Developer Response

Acknowledged, planned to normalize boost of all positions of the same size per address in a future update.

Medium Findings

1. Medium - Treasury fees can be bypassed with repeated calls to `notifyRewardAmount()`

Technical Details

Protocol fees are collected in the `pushFees` modifier when `notifyRewardAmount()` in GaugeV2.sol is called, which can be called by anyone at any time. FeeCollector.sol contract's `collectProtocolFees()` collects the pool's protocol fees and then uses the amount retrieved from the pool to calculate the treasury fee amount.

Impact

Medium. Given the low cost of gas on Arbitrum, and the fact that an amount of 0 can be passed to `notifyRewardAmount()`, the protocol fee of a pool can be kept sufficiently low such that `amount0Treasury` and `amount1Treasury` are always 0.

Recommendation

Round the `amount0Treasury` and `amount1Treasury` divisions up, so at least 1 unit of the tokens is paid to the treasury when the amount is low enough to round down to 0.

Developer Response

Acknowledged, value of amounts would have to be in the weis or very small to be possible, which gas fees would supercede.

2. Medium - `boostAmount` stays attached to a position after ownership is lost

Technical Details

When a `veRamTokenId` is attached to a `position` there is a check to ensure that the `msg.sender` `isApprovedOrOwner` of the `veRamTokenId`. If this check passes, the balance of the NFT is attached to the `boostedPosition`. However, if the ownership of the `veRamTokenId` changes, or the position owner is no longer approved, this `veRamAmount` stays attached to the `boostedPosition` indefinitely unless the `veRamTokenId` is explicitly changed, or the liquidity of the position goes to 0.

Impact

Medium. Coupling this attack with the ability to attach multiple positions to the same `veRamTokenId` an attacker can gain ownership of a new `veRamTokenId` and attach multiple positions to it, then relinquish the `veRamTokenId` ownership repeatedly ad infinitum.

Developer Response

Acknowledged, expected behavior that was kept to allow further layers to build on-top to allow seamless transferring. Due to the nature of our rewarding system promoting tighter ticks, this would not be much of an issue as the position will likely need to be withdrawn and re-created to maintain any benefit. Another mitigation is that the boost system operates on epoch-long cycles. For a user to keep the boost, they would need to re-attach at the start of each epoch– which if this transferring of ownership occured, would not be feasible.

3. Medium - Minting, burning, and minting yields higher `boostAmount` than minting

Technical Details

When minting a new position, the pool's token balances are used to determine `hypotheticalLiquidity` and therefore the `boostedLiquidityCap`, `veRamBoostAvailable`, and `positionBoostUsedRatio`.

However, when a position is burned, the tokens continue to accrue to the pool until `collect()` is called by the burner. This means that even though the tokens may be allocated to the burner, and they can retrieve them at any time, they continue to contribute to the `boostedLiquidityCap`, `veRamBoostAvailable`, and `positionBoostUsedRatio` variables when positions are updated. This means that `boostedLiquidityCap` and `veRamBoostAvailable` can be inflated, and `positionBoostUsedRatio` deflated, to any value an attacker would like without supplying additional liquidity.

See this POC illustrating the point in code:

```
contract AttachTest is Test {

    RamsesV2Pool _pool;
    RamsesV2Factory _factory;
    MockVoter _voter;
    MockNFPManager _nfpManager;
    MockVotingEscrow _veRam;
    MockERC20 _token0;
    MockERC20 _token1;


    function setUp() public {
```

```
        _pool = new RamsesV2Pool();
        _factory = new RamsesV2Factory();
        _voter = new MockVoter();
        _nfpManager = new MockNFPManager();
        _veRam = new MockVotingEscrow();
        _token0 = new MockERC20("0", "token_0", 18);
        _token1 = new MockERC20("1", "token_1", 18);

        deal(address(_token0), address(this), type(uint256).max);
        deal(address(_token1), address(this), type(uint256).max);

        vm.label(address(_pool), "_pool");
        vm.label(address(_token0), "_token0");
        vm.label(address(_token1), "_token1");
        vm.label(address(_factory), "_factory");
        vm.label(address(_veRam), "veRam");
        vm.label(address(_nfpManager), "_nfpManager");
        vm.label(address(_voter), "_voter");

        _factory.initialize(address(_nfpManager), address(_veRam), address(_voter),
address(0));
        _pool.initialize(address(_factory), address(_nfpManager), address(_veRam),
address(_voter), address(_token0), address(_token1), 10_000, 200);
        _pool.initialize(3934959759011022346558 29);
    }

    function tickSpacingToMaxLiquidityPerTick(int24 tickSpacing) public pure returns
(uint128) {
        int24 minTick = (TickMath.MIN_TICK / tickSpacing) * tickSpacing;
        int24 maxTick = (TickMath.MAX_TICK / tickSpacing) * tickSpacing;
        uint24 numTicks = uint24((maxTick - minTick) / tickSpacing) + 1;
        return type(uint128).max / numTicks;
    }

    function testBurnAndMint() public {
        uint256 period = block.timestamp / (1 weeks);
```

```
        (
            uint160 sqrtPriceX96,
            int24 tick,
            uint16 observationIndex,
            uint16 observationCardinality,
            uint16 observationCardinalityNext,
            uint8 feeProtocol,
            bool unlocked
        ) = _pool.slot0();
        int24 tickLower = tick - 132;
        int24 tickUpper = tick + 68;
        _pool.mint(address(this), 0,  tickLower, tickUpper,
tickSpacingToMaxLiquidityPerTick(200), 1, "");
        (uint128 boostAmountPreBurn,,,) = _pool.boostInfos(period,
positionHash(address(this), 0, tickLower, tickUpper));
        _pool.burn(0, tickLower, tickUpper, tickSpacingToMaxLiquidityPerTick(200), 1);
        _pool.mint(address(this), 0,  tickLower, tickUpper,
tickSpacingToMaxLiquidityPerTick(200), 1, "");
        (uint128 boostAmountPostBurn,,,) = _pool.boostInfos(period,
positionHash(address(this), 0, tickLower, tickUpper));
        assertGt(uint(boostAmountPostBurn), uint(boostAmountPreBurn));
    }
}
```

Note that the pool in the POC has been initialized similarly to the deployed DEUS/USDC pool which can be found here. Also, note that the `MockVotingEscrow` contract has been hard coded to return the same `balanceOfNFT()` (the balance taken from the deployed veRAM contract for `tokenId` 1 at the time of the test) and `isApproverOrOwner()` for ease of testing.

```
    function isApprovedOrOwner(address, uint256) external override view returns (bool) {
        return true;
    }

    function balanceOfNFT(uint256) external override view returns (uint256) {
```

```
        return 214874691303532057164742l4;
    }
```

Impact

Medium. An attacker can bypass the `boostedLiquidityCap` and `veRamBoostAvailable` limits.

Recommendation

Maintain separate token balance state variables that should be used by the boosted calculations by subtracting the tokens owed to position burners whenever a position is burned. You could also send the tokens to the burner at the time they are burned, rather than maintaining the balance in the pool.

Developer Response

Acknowledged, we plan on making a `poke()` function so users can punish bad actors if they game the system this way.

`poke()` will use current balances in the pool so if the attacker withdrew their tokens then it'll lower their boost. If the attacker leaves the withdrawable tokens in the pool then everyone else will be getting enlarged boost, negating the enlarged boost the attacker has.

4. Medium - Incorrect math in `left()`

`GaugeV2.left()` has a mistake in the logic to calculate the remaining time left of the current period.

Technical Details

The line calculating `elapsedTime` calculates the difference between `_blockTimestamp()`, with units of seconds, and `period`, with units of weeks. Subtracting weeks from seconds will result in a nonsensical result, so the two values must have the same units before the difference is calculated.

Impact

Medium. A difference of two values is calculated when they have mismatched units, making the result incorrect.

Recommendation

Modify `left()` as follows so that the two values used to calculate `elapsedTime` both have units of seconds:

```
function left(address token) external view override returns (uint256) {
        uint256 period = _blockTimestamp() / WEEK;
-       uint256 elapsedTime = _blockTimestamp() - period;
+       uint256 elapsedTime = _blockTimestamp() - (period * WEEK);
        return (tokenTotalSupplyByPeriod[period][token] * elapsedTime) / WEEK;
}
```

Developer Response

Acknowledged, fixed in 7285a1216e5aa13906e2681f20a5451e52a742c3.

Low Findings

1. Low - The first mint in a pool gets 0 `boostAmount` regardless of `veRamTokenId` or liquidity supplied

Technical Details

The `boostedLiquidityCap` is based on the `hypotheticalLiquidity` which is based on the pool's token balances. When a pool is first deployed, it has no liquidity, so the first minter will receive no `boostedLiquidity` regardless of the size of their mint or `veRamTokenId`'s balance. This is particularly acute if the minter is minting a large position as is illustrated in the POC below.

See this POC for an illustration of the point in code:

```
    function testFirstMint() public {
        uint256 period = block.timestamp / (1 weeks);
        (
            ,
            int24 tick,
            ,
            ,
            ,
            ,
        ) = _pool.slot0();
        int24 tickLower = tick - 132;
        int24 tickUpper = tick + 68;
        _pool.mint(address(this), 0,  tickLower, tickUpper,
```

```
tickSpacingToMaxLiquidityPerTick(200) - 100, 1, "");
        (uint128 boostAmountFirstMint,,,) = _pool.boostInfos(period,
positionHash(address(this), 0, tickLower, tickUpper));
        assertEq(uint256(boostAmountFirstMint), 0);
        _pool.mint(address(this), 1,  tickLower, tickUpper, 100, 2, ""); // This could be
a new minter, however we're using a different index and veRamTokenId for illustration
purposes.
        (uint128 boostAmountSecondMintDifferentPosition,,,) = _pool.boostInfos(period,
positionHash(address(this), 1, tickLower, tickUpper));
        assertGt(uint256(boostAmountSecondMintDifferentPosition), 0);
    }
```

Impact

Low. This should only affect a pool on the first mint. Afterward, there should be liquidity in the pool.

Developer Response

Acknowledged.

2. Low - `GaugeV2.getPeriodReward()` can skip `lastClaimByToken` for a valid claim

`lastClaimByToken` is used for the last time (< current period) the user claimed rewards. Still, it can be possible to claim the reward for a period without changing the value of `lastClaimByToken`.

Technical Details

In function, `GaugeV2.getPeriodReward()`, consider the following:

- User wants to claim the reward for period = 13, current period = 14, and `lastClaimByToken` was made for all the tokens in period 10.
- That means the condition comes out to be `10 < period < 13.`

```
            if (
                period > lastClaimByToken[tokens[i]][_positionHash] &&
                period < _blockTimestamp() / WEEK - 1
            )
```

- The period for which the rewards were claimed doesn't lie in the range. Hence, it skips the check and then moves on to the `_getReward()` call, which lets the user claim a reward for period 13, and `lastClaimByToken` remained 10.

Well, it should be able to set the `lastClaim` to that period as it has been passed, and the user can claim the reward for that period later in the function.

Impact

Low.

Recommendation

It's sufficient to use `period < _blockTimestamp() / WEEK` to confirm the `lastClaim` can be made for past period.

Developer Response

Acknowledged.

3. Low - The `rewards` list can grow unbounded in GaugeV2.sol

Technical Details

`notifyRewardAmount()` is an external function, with no access control modifier, which means anyone can call `notifyRewardAmount()`. If a token has not been seen before, it is added to the `isReward` mapping and pushed into the `rewards` list. This means an attacker could fill the list with a large number of random token addresses. Elsewhere in GaugeV2.sol, such as `getPeriodReward()`, tokens are passed into the function such that the entire reward list is not iterated through. However, this is not the case in the RamsesRewards.sol contract's `addressEarnedCl()` function, where the entire reward list is iterated through.

Impact

Low. Currently, the Gauge's reward list is used in the `addressEarnedCl()` function of RamsesRewards.sol, which is a `view` function, so the impact is not high.

Recommendation

Put a length limit on the reward list, and/or allow the owner to remove reward tokens from the list, and/or access control the `notifyRewardAmount()` function in GaugeV2.sol.

Developer Response

Acknowledged.

4. Low - `createCLGauge()` code doesn't match comment

A comment in `Voter.createCLGauge()` does not match the code.

Technical Details

This comment and a line of code earlier in the function directly conflict with one another.

```
require(_pool != address(0), "no pool");
...
// gov can create for any pool, even non-Ramses pairs
```

By requiring `_pool != address(0)` value of true, governance is limited to only adding gauges to valid RAMSES pairs. This is in contrast to `createGauge()` for Ramses v1, where the equivalent check requiring `isPair` to be true is only applied to a non-governor `msg.sender`.

Impact

Low. Governor does not have the elevated capabilities for adding gauges like RAMSES v1, so the comment or the code is incorrect.

Recommendation

Move the `require(_pool != address(0), "no pool");` line of code into the if branch where `msg.sender != governor`.

Developer Response

Updated the comment in 95192c840e8b2753b821c2f9f29294b12410d1aa

5. Low - Inconsistent first period in GaugeV2

The first period in GaugeV2 will not consistent of a full week of time unless the contract is initialized perfectly on time, which is unlikely. Instead, the first period will have less than 1 week remaining as calculated by the function `left()` starting from the moment of initialization.

Technical Details

The `firstPeriod` variable is set during initialization as:

```
firstPeriod = _blockTimestamp() / WEEK;
```

This value will be rounded down. The end result of this value rounding down is the same whether the week of `firstPeriod` is 5% complete or 95% complete during initialization. This

rounding down will artificially shorten the time period for the `firstPeriod`. This may skew the rewards per second during `firstPeriod` compared to future periods.

Impact

Low. `firstPeriod` duration will be inconsistent compared to later periods.

Recommendation

Add logic to account for the reduced `firstPeriod` duration. Or add a comment in the initialization function acknowledging this variation and explaining why it is acceptable.

Developer Response

Acknowledged.

6. Low - Errors in `left()`

`Gauge.left()` has several minor errors that should be fixed.

Technical Details

1. The NatSpec for `left()` references a function that does not exist, `getTokenTotalSupplyByPeriod()`. The NatSpec should be updated to remove this reference.
2. The name of the function `left()` is misleading. The return value sees the `elapsedTime` value in the numerator, not the remaining time. This means that the return value of `left()` increases as the period approaches the end of the period, when the function name indicates that the opposite should be true.

Impact

Low. `left()` function is misleading and should be updated.

Recommendation

Modify `left()` and the associated comments to fix these errors.

Developer Response

Acknowledged, fixed in 7285a1216e5aa13906e2681f20a5451e52a742c3.

7. Low - No validation of `pool` address in `createGauge()`

`createGauge()` has an address value function argument that should correlate to the value of a RAMSES pool contract address. This address is never verified to be related to RAMSES in any way, and because `createGauge()` is permissionless, anyone can call the function and provide an

address of their choice. This may cause problems later with RAMSES contracts interacting with an untrusted external contracts.

Technical Details

Like other contracts in RAMSES, RamsesV2GaugeFactory.sol is loosely based on UniswapV3Factory.sol, with the implementation of `createGauge()` based on `createPool()`. In Uniswap, anyone can create a pool between two ERC20 tokens. It is unclear whether the same should be true of `createGauge()`, but there is no access control enforced on this function. The `pool` address function argument is not verified as a RAMSES pool before it is passed to `_deploy()`, which also does not validate the `pool` address value. The result is that a gauge will exist with a `pool` address that is not a pool, and a malicious party can create a contract that implements the same interface as a RAMSES pool but does not necessarily perform the same actions. There are external calls to the `pool` address in GaugeV2.sol (such as 1, 2, 3) that may be leveraged into an attack vector, though the specifics have not been fully worked out yet.

Impact

Low. RAMSES will store an untrusted address and make external calls to this address.

Recommendation

Validate the `pool` function argument before storing it. Modify the code by creating a `ramv2Factory` variable to store the RamsesV2Factory address and then modify `createGauge()` as follows:

```
function createGauge(
-       address pool
+       address token0,
+       address token1,
+       uint24 fee
) external override returns (address gauge) {
+       require(ramv2Factory.getPool[token0][token1][fee] != address(0), "PE");
        require(getGauge[pool] == address(0), "GE");
        gauge = _deploy(voter, nfpManager, feeCollector, pool);
        getGauge[pool] = gauge;
        emit GaugeCreated(pool, gauge);
}
```

Developer Response

Acknowledged, fixed in 7285a1216e5aa13906e2681f20a5451e52a742c3.

Gas Saving Findings

1. Gas - v1 Gauge.sol `poke()` should cache length before loop

While out of scope, there is a gas savings in RAMSES v1 `Gauge.poke()`.

Technical Details

`Gauge.poke()` should cache `rewards.length` before the for loop, then reference the cache length value in the loop, to achieve gas savings. Otherwise the length of the `rewards` state variable is checked on each iteration of the loop.

This finding was automatically identified by slither.

Impact

Gas savings.

Recommendation

Modify the for loop to cache the length value before the loop.

```
- for (uint256 i; i < rewards.length; ++i) {
uint256 rlength = rewards.length;
+ for (uint256 i; i < rlength; ++i) {
```

2. Gas - `_find_time_user_epoch()` loop iterations can be reduced

`VotingEscrow._find_time_user_epoch()` has binary search logic similar to that found in VotingEscrow.vy in Curve Finance. The for loop in this function can be made more efficient.

Technical Details

VotingEscrow.vy in Curve has a function `find_block_epoch()` which implements binary search using a for loop that repeats 128 times. RAMSES has a similar `VotingEscrow._find_time_user_epoch()` function that implements binary search using a for loop that repeats 256 times. The RAMSES `VotingEscrow._find_time_user_epoch()` implementation is in contrast to the RAMSES implementation of `VotingEscrow._balanceOfAtNFT()`, which loops 128 times like Curve. There is no need for `VotingEscrow._find_time_user_epoch()` to loop 256 times, it can stop at 128.

Impact

Gas savings.

Recommendation

Modify the for loop in `_find_time_user_epoch()` to loop only 128 times.

3. Gas - Move logic into if statement

A line in `Voter._updateFor()` can be moved into the branching logic to save gas when a specific branch is skipped.

Technical Details

`_supplied` is only needed in the if branch, so it can be set inside that branch.

```
function _updateFor(address _gauge) internal {
        address _pool = poolForGauge[_gauge];
-       uint256 _supplied = weights[_pool];
        uint256 _supplyIndex = supplyIndex[_gauge];

        // only new pools will have 0 _supplyIndex
        if (_supplyIndex > 0) {
+               uint256 _supplied = weights[_pool];
                uint256 _index = index; // get global index0 for accumulated distro
```

Impact

Gas savings.

Recommendation

Move variable declaration into branching logic.

4. Gas - `periodSecondsPerBoostedLiquidityOutsideBeforeX128` and `periodSecondsPerLiquidityOutsideBeforeX128` are 0 and can be removed from subtraction

Technical Details

In `cross()` `periodSecondsPerLiquidityOutsideBeforeX128` and `periodSecondsPerBoostedLiquidityOutsideBeforeX128` must be 0 for the if logic to be executed. In this case, their subtraction does nothing to contribute to the final `periodSecondsPerLiquidityOutsideX128` and `periodSecondsPerBoostedLiquidityOutsideX128` values.

Impact

Gas savings.

Recommendation

Remove the variables from the subtraction.

5. Gas - `ERC721PermitUpgradeable` inherits from `Initializable` twice

There is duplicate inheritance in `ERC721PermitUpgradeable`, meaning one of these instances can be removed.

Technical Details

`ERC721PermitUpgradeable` inherits from `Initializable` explicitly. However, `ERC721Upgradeable` already inherits from `Initializable`.

Impact

Gas savings.

Recommendation

Remove the explicit `Initializable` inheritance.

6. Gas - `positionLiquidity - params.liquidity` is calculated twice

Duplicate calculation of a value can be optimized to save gas.

Technical Details

`position.liquidity` is stored as `positionLiquidity - params.liquidity`. This is recomputed to check if the liquidity difference is 0 and unset the veRamTokenId.

Impact

Gas savings.

Recommendation

Check if `position.liquidity` is 0 rather than re-computing `positionLiquidity - params.liquidity`.

7. Gas - Gas can be saved calling a different `periodEarned()` in GaugeV2.sol

Two versions of the `periodEarned()` function are available and calling the other version of the function can save gas.

Technical Details

Currently, in `earned()` `periodEarned(period, token, tokenId)` is called to accumulate the reward. This `periodEarned()` function re-computes the `tickLower` and `tickUpper` which are already known at the time `periodEarned()` is called in `earned()`.

Impact

Gas savings.

Recommendation

Use the `periodEarned()` that includes `tickLower` and `tickUpper` as parameters in `earned()` when accumulating the rewards in `earned()`.

Informational Findings

1. Informational - Restrict reward claim for current epoch

The current implementation of `GaugeV2` allows the user to claim the reward for the current epoch using the function `getReward()` which does an internal call to `_getAllRewards` which lets the user claim the reward for the current epoch.

Technical Details

Since all the important (rewards deciding) variables for the current epoch get set at the start of the next epoch, it is crucial to not let the user claim the reward for the current epoch. Additionally, the reward that the user is claiming for the current week turns out to be an arbitrary value calculated using a fixed value of `boostedInRange` as `1 WEEK`. Even if the user claims the reward for the current week, his/her rewards are still accumulating which the user can't claim as those rewards become available in the next epoch. Then it is better to restrict users to claim rewards for the current epoch.

Impact

Informational.

Recommendation

In `_getAllRewards()`:

```
    function _getAllRewards(
        address owner,
        uint256 index,
        int24 tickLower,
```

```
        int24 tickUpper,
        address[] memory tokens,
        address receiver
    ) internal {
        bytes32 _positionHash = positionHash(
            owner,
            index,
            tickLower,
            tickUpper
        );
        uint256 currentPeriod = _blockTimestamp() / WEEK;
        uint256 lastClaim;
        for (uint256 i = 0; i < tokens.length; ++i) {
            lastClaim = Math.max(
                lastClaimByToken[tokens[i]][_positionHash],
                firstPeriod
            );

            for (
                uint256 period = lastClaim;
-                period <= currentPeriod;
+               period < currentPeriod;
                ++period
            ) {
                _getReward(
                    period,
                    tokens[i],
                    owner,
                    index,
                    tickLower,
                    tickUpper,
                    _positionHash,
                    receiver
                );
            }
            lastClaimByToken[tokens[i]][_positionHash] = currentPeriod - 1;
```

```
        }
    }
```

2. Informational - `initialize` should emit event for variable set

As `initialize` is used to set the important variables in the beacon proxy pattern, it doesn't emit an event for setting up those important variables.

Technical Details

Ideally, the following should trigger events.

- `FeeCollector.initialize()` should emit `TreasuryChanged`.
- `RamsesV2Factory.initialize()` should emit `ImplementationChanged`

Impact

Informational.

Recommendation

Add event emission event for setting them for the first time, as Uniswap v3 follows this practice.

3. Informational - `_switchAttachment` shouldn't allow switch if `position.liquidity == 0`

One of the protocol invariants is to detach `veRamTokenId` (setting it to 0) from pool/nfpManager if the liquidity of the position becomes zero, but `_switchAttachment` could be used to attach a `veRamTokenId` even if the position has no liquidity.

Technical Details

Once a user decreases all the liquidity from its `nfpManager` position, the `veRamTokenId` from both the pool and `nfpManager` position gets detached (becomes 0). But if the user calls `_switchAttachment` it will set `veRamTokenId` in `nfpManager`, but in the pool, the user `attachedVeRamId` remains zero. (`nfpManager.switchAttachment()` -> `pool.burn()` -> `_updateLiquidity()` which only updates `veRamTokenId`, if `newLiquidity > 0`).

This can make the pool and nfpManager contract out of sync for a user position.

Impact

Informational.

Recommendation

```
    function switchAttachment(
        uint256 tokenId,
        uint256 veRamTokenId
    ) external override isAuthorizedForToken(tokenId) {
        if (veRamTokenId != 0) {
            require(
                IVotingEscrow(veRam()).isApprovedOrOwner(
                    msg.sender,
                    veRamTokenId
                ),
                "veRam not approved"
            );
        }

        Position storage position = _positions[tokenId];

+       require(position.liquidity > 0, "blah");
        PoolAddress.PoolKey memory poolKey = _poolIdToPoolKey[position.poolId];

        IRamsesV2Pool pool = IRamsesV2Pool(
            PoolAddress.computeAddress(factory, poolKey)
        );

        emit SwitchAttachment(tokenId, position.veRamTokenId, veRamTokenId);

        position.veRamTokenId = veRamTokenId;

        pool.burn(
            tokenId,
            position.tickLower,
            position.tickUpper,
            0,
            veRamTokenId
```

```
        );
    }
```

4. Informational - Incorrect NatSpec comment in `ProtocolActions` Library

Event `CollectProtocol` has an incorrect comment.

Technical Details

`CollectProtocol` event is emitted when `feeCollector` calls the `collectProtocol` function of RamsesV2Pool.sol, as the function has a modifier `onlyFeeCollector` which means it can only be called by `feeCollector`.

Impact

Informational.

Recommendation

```
-    /// @notice Emitted when the collected protocol fees are withdrawn by the factory owner
+  /// @notice Emitted when the collected protocol fees are withdrawn by the fee collector
    /// @param sender The address that collects the protocol fees
    /// @param recipient The address that receives the collected protocol fees
    /// @param amount0 The amount of token0 protocol fees that is withdrawn
    /// @param amount0 The amount of token1 protocol fees that is withdrawn
    event CollectProtocol(address indexed sender, address indexed recipient, uint128 amount0, uint128 amount1);
```

5. Informational - Revert doesn't contain error message

Failed operations in the smart contracts in scope are reverted with error messages provided in the `require` statement, making it easier to debug the reason for the failure of the transaction.

Technical Details

Revert statement reverts without error message in the following functions.

- RamsesV2Pool.sol:

    - `onlyFeeCollector()`
    - `mint()`
- RamsesV2Factory.sol:
    - `enableFeeAmount()`
- NonfungiblePositionManager.sol:
    - `tokenURI()`
    - `decreaseLiquidity()`
- SwapRouter.sol:
    - `ramsesV2SwapCallback()`
    - `exactOutputInternal()`

Impact

Informational.

Recommendation

Use revert string, to make transaction reverts easily detectable.

6. Informational - inconsistency in preventing implementation initialization

Technical Details

There needs to be more consistent behavior in the constructor of implementation contracts. The following contracts use `_disableInitializers();` to disable initialization of implementation (which is good):

- GuageV2.sol
- FeeCollector.sol
- RamsesLens.sol

But other contracts follow `constructor() initializer {}` (which does the same thing, but it is recommended to use `_disableInitializers` for implementations)

Impact

Informational.

Recommendation

Use consistent behavior to prevent initialization of implementation. Also, `_disableInitializers` is more gas efficient.

7. Informational - Imports are unused

Technical Details

Various imports throughout the codebase are not used and can be removed.

- The IWETH9.sol import in SwapRouter.sol.
- The SqrtPriceMath.sol, IRamsesV2PoolDeployer.sol, IVotingEscrow.sol, and IVoter.sol imports in RamsesV2Pool.sol.
- RamsesPool.sol in RamsesV2Factory.sol.
- IGaugeV2.sol in RamsesV2GaugeFactory.sol.

Impact

Informational

Recommendation

Remove the imports.

8. Informational - Contract without proxy can be simplified

RamsesDeployer.sol for RAMSES v2 is not deployed behind a proxy. This is in contrast to the nearly identical RamsesDeployer.sol for RAMSES v1, which is deployed behind a contract. The v2 contract could be simplified because it contains proxy artifacts.

Technical Details

Several proxy artifacts in RamsesDeployer.sol can be modified because it is not behind a proxy:

1. The file imported from @openzeppelin/contracts-upgradeable can be changed to the @openzeppelin/contracts version of the file.
2. Initializer is not necessary.
3. `initialize()` can be replaced by the contract constructor.

Impact

Informational.

Recommendation

Remove proxy artifacts when a contract is not deployed behind a proxy.

9. Informational - Whitelisted tokens for gauges cannot be removed

`Voter.whitelist()` allows specific tokens to be whitelisted so that non-governance users can create a gauge for pools with whitelisted tokens. There is no way to reverse the process and remove a token from the whitelist. The goal of this may be to decrease the power of the multisig over the protocol, but if there is a vulnerability in a whitelisted token, the lack of a `blacklist()` function to remove a token from the whitelist may make it more difficult to protect the protocol from certain risks.

Technical Details

There is no way to reverse the action of the `whitelist()` operation in Voter.sol. If there is a token that should be removed from the whitelist, the Voter contract (which is deployed behind a proxy) may need to be replaced. Adding a function to reverse the whitelisting of tokens can improve the mitigation of future threats if a trusted token contract is maliciously or incompatibly upgraded.

Impact

Informational.

Recommendation

Add a way to reverse the action of `whitelist()`. This function could be accessible to only the `emergencyCouncil` role, which may have a higher threshold of activation to the `governor` role.

10. Informational - Emergency council identical to governor in Voter.sol

The RAMSES docs have a section titled “The Path Towards Decentralization”, but in summary the docs say the protocol is working on this aspect of the protocol. One example is `governor` and the `emergencyCouncil` of Voter.sol are identical values, with the RAMSES multisig assigned to those two state variables. This choice of using the same multisig for the two values is different from other Solidly forks.

Technical Details

Voter.sol has a `governor` role and an `emergencyCouncil` role. In the on-chain contract, the same RAMSES multisig is set for these two different roles. In other Solidly fork implementations with the `emergencyCouncil` role, such as Velodrome on Optimism and PearlFi on Polygon, these roles are assigned to different multisigs. Specifically, the `emergencyCouncil` multisig has a higher threshold than the governance multisig, suggesting a design choice where controlling the uptime of the protocol requires cooperation from more signers than adding gauges to the protocol.

This is a design choice and RAMSES may have decided that the current approach is best, but without clear documentation on this choice, the current settings make the RAMSES multisig a centralized source of power over the full protocol.

Impact

Informational.

Recommendation

Consider creating a new `emergencyCouncil` multisig for the protocol with a higher signer threshold compared to the main multisig.

11. Informational - On-chain contracts using different library versions

Some of the on-chain contracts are using older versions of library code. These differences are minor and unlikely to cause issues, but using consistent code across all contracts in the protocol could remove some risk by reducing the amount of unique code deployed.

Technical Details

Some of the on-chain code does not match the contract versions reviewed during this audit. Specifically, the on-chain Oracle.sol deployment at 0xf70c9C4F6281C0750d68da8878894b1235cb6020 has some minor differences in 3 dependencies, States.sol, LiquidityMath.sol, and Tick.sol, compared to the code reviewed in this audit.

Similarly, Tick.sol deployed at 0xc989D669831Cd5258369CB0Dce7752CbfE7303E8 and ProtocolActions.sol deployed at 0xa67f82621540017a679153423CA0B8a1b4518B49 have different versions of the State.sol library dependency compared to the version of State.sol in this audit.

Impact

Informational.

Recommendation

Consider upgrading all code to use the same version of libraries.

12. Informational - Reduce rounding error inaccuracy

`GaugeV2.rewardRate()` includes two division operations that can be combined into one. This would reduce the inaccuracy due to two rounding down operations.

Technical Details

The two division operations can be reduced to one to minimize error created by the rounding down inherent in a solidity division operation.

```
function rewardRate(address token) external view returns (uint256) {
        uint256 period = _blockTimestamp() / WEEK;
-       return (tokenTotalSupplyByPeriod[period][token] * 4) / 10 / WEEK;
+       return (tokenTotalSupplyByPeriod[period][token] * 4) / (10 * WEEK);
}
```

Impact

Informational.

Recommendation

Remove a division operation to reduce rounding inaccuracy.

Developer Response

Acknowledged, fixed in 7285a1216e5aa13906e2681f20a5451e52a742c3.

13. Informational - Beacon proxy inconsistency

There is a slight inconsistency in how the beacon proxy is configured for pools and how they are configured for gauges.

Technical Details

RamsesV2PoolDeployer uses the value `keccak256(abi.encode())` for the beacon proxy creation while RamsesV2GaugeDeployer uses `keccak256(abi.encodePacked())`. These contracts perform almost the exact same task, so the same code logic should be used.

Impact

Informational.

Recommendation

Use a consistent ABI packing for similar functions. Consider using `abi.encode()` over `abi.encodePacked()` to avoid collisions, even though there are no dynamic types provided as input to the ABI encoding currently.

14. Informational - `STATES_SLOT` can accidentally match a known pre-image

Technical Details

By calculating `STATES_SLOT = keccak256("states.storage")` using a string value, the first pre-image of `STATES_SLOT` is known ("states.storage"), which makes it accidentally match the known pre-image ("states.storage"). Openzeppelin and all proxy standards derive their custom storage slot in this way. Ref

Impact

Informational.

Recommendation

- It is general practice to refrain from using pre-images while fixing important storage slots.

```
bytes32 STATES_SLOT = keccak256("states.storage") - 1
```

- Make `STATES_SLOT` a constant variable.

By doing so, the first pre-image of `STATES_SLOT` becomes unknown, enhancing the storage slot's security.

15. Informational - First period has default zero value for `startTick` and `previousPeriod`

`RamsesV2Pool._advancePeriod()` updates the pool state values when advancing to the next period. `startTick` and `previousPeriod` are values in the pool state that are set at the start of a new period, but because these values are only set when there is a previous period, the first period does not have these values set. Therefore, the first period will store the default zero value for `startTick` and `previousPeriod`, which makes handling these values potentially more difficult for the special case of the first period.

Technical Details

The implementation of `RamsesV2Pool._advancePeriod()` primarily updates the values for `states.periods[_lastPeriod]`, or the state of the last period. Only one line of code updates the current period value. Because `_advancePeriod()` is first called in `initialize()`, the initial `_lastPeriod` value of zero will be meaningless, but it will have no values set for `startTick` and `previousPeriod`, so these values will remain zero. The first real period is the 1st period (not the 0th period). Any loop that iterates through the period states should consider this, but there is no comment to clarify this is how the period values are handled.

Impact

Informational.

Recommendation

Clarify that the 0 index of `states.periods[]` should be ignored and the first period has an index of 1. This should be documented in the NatSpec for `_advancePeriod()`.

16. Informational - Forked Uniswap v3 code uses OZ 3.4.1 not 3.4.2

Uniswap v3 Periphery depends on OpenZeppelin contracts version 3.4.2-solc-0.7. The RAMSES `v2-periphery/` directory, which is a direct fork of the Uniswap code, uses OZ version 3.4.1.

Technical Details

The changes made between 3.4.2 and 3.4.1 are minimal, which just one commit made in `TimelockController`, which is not directly used in the `v2-periphery/` directory. However, if the goal is to maintain maximum similarity to the original Uniswap, this dependency on OpenZeppelin in the `v2-periphery/` directory should be upgraded to OZ 3.4.2.

Impact

Informational.

Recommendation

Upgrade v2-periphery files to use OZ 3.4.2.

17. Informational - Improve naming choices

There are several name choices in the codebase that could be improved.

Technical Details

Several naming choices that could be improved include:

1. RamsesV2Pool.sol has two functions named "initialize". The second function sets the initial price for the pool and may be better named "setInitialPoolPrice" to avoid confusion. The function name "initialize" is generally used for a single function in an implementation contract sitting behind a proxy. This function can only be called once and sets initial values similar to how a constructor would work in a non-proxy context. But in RamsesV2Pool.sol, the second initialize function can be called more than once because it lacks an `initializer` modifier.
2. PeripheryUpgradeable.sol is a renamed version of PeripheryImmutableState.sol in Uniswap v3. The interface for PeripheryUpgradeable.sol still has the original name of IPeripheryImmutableState.sol, which does not match the contract implementing the interface file. PeripheryUpgradeable.sol is a modified version of Uniswap's

PeripheryImmutableState.sol. But the interface that is implemented in the RAMSES file is not renamed to match the contract name. Instead, the original IPeripheryImmutableState interface name remains.

3 Although out of scope, the `_gauges` function argument in `Voter.initialize()` would be better named `_gaugefactory`, because the variable name `_gauges` is used elsewhere in the contract to refer to a different variable.

Impact

Informational.

Recommendation

Improve naming choices in several places.

18. Informational - Remove library functions that are never used

At least one library function does not get called in the codebase and can be removed.

Technical Details

`checkTicks()` is declared at the end of Tick.sol but is never called. The only place where the code acknowledges this function is a comment in `collect()` with the word "checkTicks". `checkTicks()` could be called in `RamsesV2Pool.snapshotCumulativesInside()` to replace the existing tick checks, but this may only make sense if `checkTicks()` gets called in more than one location.

Impact

Informational.

Recommendation

Remove unused functions and comments referencing such functions.

19. Informational - RamsesV2Pool view functions don't return all struct values

The design may be intentional, but several view functions in `RamsesV2Pool` that only have the purpose of returning custom struct values omit some values stored in the struct.

Technical Details

One specific example of this is `ticks()` that does not return:

1 `cleanUnusedSlot`

```
2 cleanUnusedSlot2
3 periodSecondsPerLiquidityOutsideX128
4 periodSecondsPerBoostedLiquidityOutsideX128
```

Another example is `boostInfos(uint256, bytes32)` which returns values from a `BoostInfo` struct but does not return `secondsPerLiquidityPeriodStartX128` or `secondsPerBoostedLiquidityPeriodStartX128`.

A third example is `observations()` which returns all values from the `Observation` struct except for `boostedInRange`.

Impact

Informational.

Recommendation

When the function's only goal is to return a struct, consider documenting the intentional omission of certain values from the return values to avoid assumptions that the omissions may be a mistake.

Final remarks

The RAMSES team was made aware of the Velodrome Spearbit audit findings which were also applicable to RAMSES at the frozen commit. However, these findings were left out of this report for brevity's sake such as to avoid duplicate reporting.

The RAMSES team should focus on adding more robust testing, particularly of the interactions between the forked code and new code, as well as fork, fuzz, and invariant testing, as many of the issues reported could have been caught with more robust unit tests. While much of the RAMSES code in this review is forked from Uniswap v3 and uses Uniswap v3 tests, the modifications to the forked code do not have any tests. For example, there were limited, or no unit tests for the new `veRamTokenId` value used in pools. The lack of unit tests in modified code and the possibility that the assumptions inherited from the forked code are not deeply understood or considered in the modified design could lead to issues in the way that the forked code is integrated with novel code.